PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 : G06F 9/28, 15/16

A1

(11) International Publication Number: WO 93/21577

(43) International Publication Date: 28 October 1993 (28.10.93)

(21) International Application Number: PCT/US93/03165

(22) International Filing Date: 2 April 1993 (02.04.93)

(30) Priority data:
857,655 9 April 1992 (09.04.92) US

(71) Applicant: ELECTRONIC ASSOCIATES, INC. [US/US]; 185 Monmouth Parkway, West Long Branch, NJ 07764-9989 (US).

(72) Inventors: HANNAUER, George ; 41 Yorkshire Drive, East Windsor, NJ 08520 (US). DOUCEUR, John, R. ; 324 Crawford Street, Shrewsbury Township, NJ 07724 (US).

(74) Agents: RATNER, Allan et al.; Ratner & Prestia, 500 N. Gulph Road, Suite 412 The Leighton Bldg., P.O. Box 980, Valley Forge, PA 19482 (US).

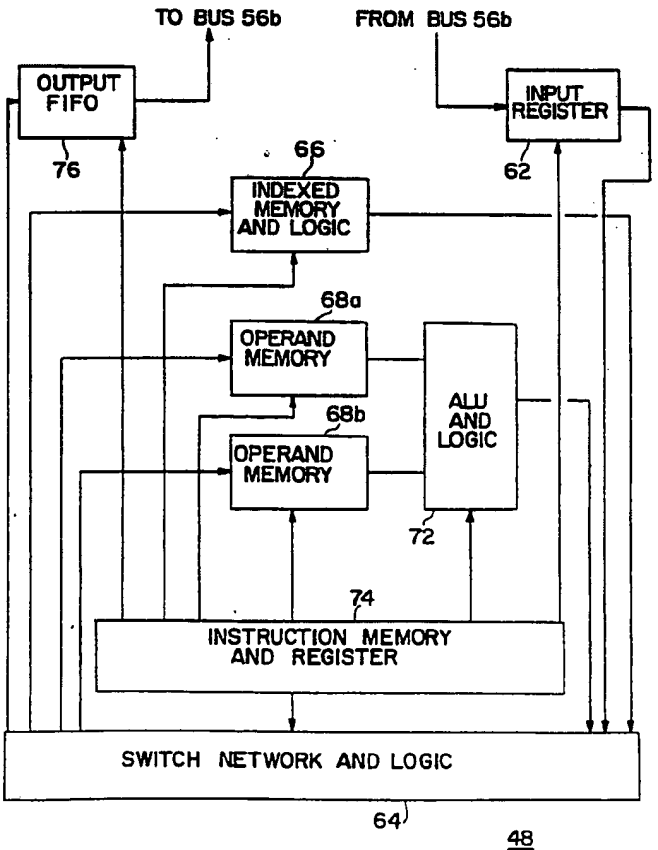
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: MULTIPROCESSOR COMPUTER SYSTEM AND METHOD FOR PARALLEL PROCESSING OF SCALAR OPERATIONS

(57) Abstract

The present invention provides a multiprocessor computer system and method for parallel processing of scalar operations. The computer system includes a single program counter for multiple arithmetic computational modules (ACMs) which are coupled in parallel by a bus (56a-c). Each ACM (48) includes an instruction memory (74), operand memories for providing operands to an ALU (72) which performs scalar operations and which generates a condition code. Additionally, each ACM (48) includes a multiplexer for selecting, based on the condition code, between the ALU (72) output, stored operands and constants and, thus, providing a data-formatted output. Also within each ACM is a switch network (64) for routing the data-formatted output to the bus or back to the ALU for subsequent scalar operations.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**FR**	France	**MR**	Mauritania
AU	Australia	**GA**	Gabon	**MW**	Malawi
BB	Barbados	**GB**	United Kingdom	**NL**	Netherlands
BE	Belgium	**GN**	Guinea	**NO**	Norway
BF	Burkina Faso	**GR**	Greece	**NZ**	New Zealand
BG	Bulgaria	**HU**	Hungary	**PL**	Poland
BJ	Benin	**IE**	Ireland	**PT**	Portugal
BR	Brazil	**IT**	Italy	**RO**	Romania
CA	Canada	**JP**	Japan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SK**	Slovak Republic
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SN**	Senegal
CM	Cameroon	**LI**	Liechtenstein	**SU**	Soviet Union
CS	Czechoslovakia	**LK**	Sri Lanka	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**MC**	Monaco	**UA**	Ukraine
DK	Denmark	**MG**	Madagascar	**US**	United States of America
ES	Spain	**ML**	Mali	**VN**	Viet Nam
FI	Finland	**MN**	Mongolia		

MULTIPROCESSOR COMPUTER SYSTEM AND METHOD FOR PARALLEL PROCESSING OF SCALAR OPERATIONS

A microfiche appendix is included in this application containing two microfiche. Microfiche number one (PHASE 1) contains 55 frames plus one test target frame, for a total of 56 frames. Microfiche number two (PHASE 2) contains 33 frames plus one test target frame for a total of 34 frames.

FIELD OF THE INVENTION

The present invention relates generally to computer systems and methods which use parallel processing

techniques and, more particularly, to multiprocessing computer systems and methods designed for parallel processing of scalar operations within simulation applications.

BACKGROUND OF THE INVENTION

Simulation is an invaluable tool for the evaluation and development of all types of systems including mechanical (e.g., robotic arm), chemical (e.g., chemical reactions), electrical (e.g., application specific integrated circuits), etc. The simulation of some systems can be a computationally intensive task entailing many scalar operations.

Attempts have been made to perform such simulations with vector machines which are efficient when they can be configured to perform an instruction on vectors (large groups) of data. However, some systems do not lend themselves to vectors of data, instead they require many scalar operations. In this type of application, vector machines are not efficient and do not benefit from their vector-oriented design.

An alternate architecture to the vector machines for these applications is the single-bus multiprocessor AD100. In its basic form, this system consists of five devices attached to a single bus: an adder, a multiplier, a data memory, a control processor, and a host interface. In this architecture, all processors communicate via the same bus which increases the probability of bus contention with consequent loss of speed. Additional drawbacks include 1) the operations are not pipelined, 2) there is no critical-path scheduling and 3) there is no provision for expanding beyond a single AD100 system.

Another parallel-processing architecture to be considered is MultiFlow, Inc.'s TRACE supercomputer. The TRACE system includes seven functional units: two integer, two floating point, two load/store (memory access) and a branch-logic controller. In addition, the TRACE architecture allows up to four such systems to be interconnected for a total of 28 functional units. However, because the TRACE system has a single memory for each 7-unit system, it is capable of only two memory references per clock cycle.

Additionally, the compiler for the TRACE system operates by using a technique called "trace compacting". This technique handles conditional jumps by guessing which branch is most likely to be taken and generating code based on the guess. Then, if a different branch is actually taken at run-time, the compiler has prepared "fix-up" code to correct the resulting error.

SUMMARY OF THE INVENTION

A multiprocessor computer system having a bus for parallel processing of scalar operations. The computer system further includes a single program counter and a plurality of arithmetic computational modules (ACMs). Each ACM includes an instruction memory responsive to the single program counter, operand memories for providing operands to an ALU which performs scalar operations and provides a condition code. Additionally, each ACM includes a multiplexer for selecting between the ALU output, stored operands and constants as a function of the condition code and providing a data-formatted output. A switch network, provided in each ACM, routes the data-formatted output to the bus or back to the operand memories and/or ALU for subsequent scalar operations.

Another aspect of the present invention is a method for transforming multi-input (more than two) operations into two-input operations such that the same result is obtained and the critical path is minimized. The method includes the steps of calculating the earliest finish time (EFT) for each variable and creating the two-input operations based on the lowest EFTs.

Another aspect of the present invention is a method for scheduling the operations to be performed by an ACM. The method comprises the steps of calculating the down-stream time (DST) for each variable and, as operations become available to be performed (due to the availability of its operands), the operations are scheduled; however, if more than one operation is ready than the one with the greatest DST is chosen.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is best understood from the following detailed description when read in connection with the accompanying drawings, in which:

Fig. 1 is a dataflow diagram of a prior art analog system;

Fig. 2A is a high-level functional block diagram of Fig. 1 implemented with analog technology;

Fig. 2B is a high-level functional block diagram of Fig. 2A implemented with digital technology;

Fig. 2C is a high-level functional block diagram of Fig. 2B implemented with digital technology and designed to time-multiplex components;

Fig. 2D is a high-level functional block diagram of Fig. 2C implemented with digital technology and designed to perform all functions by time-multiplexing;

Fig. 3 is a high-level functional block diagram of a system incorporating the present invention which employs a plurality of Arithmetic Control Modules (ACM's) in parallel;

Fig. 4 is a high-level functional block diagram of an individual processing element or arithmetic computational module (ACM);

Fig. 5 is a detailed functional block diagram of an a portion of the ACM in Fig. 4;

Fig. 6A and 6B together show a detailed functional block diagram of the ACM in Fig. 4;

Fig. 7A is a low-level dataflow diagram depicting how a typical prior art compiler schedules a task;

Fig. 7B is a low-level dataflow diagram depicting how a compiler incorporating an aspect of the present invention schedules the same task as shown in Fig. 7A; and

Fig. 8 is a flow chart depicting how a compiler incorporating an aspect of the present invention determines how to schedule tasks like the one in Fig. 7B.

DETAILED DESCRIPTION OF THE INVENTION

Evolution of Digitally-Implemented Analog Computer

A multiprocessor computer system is shown which benefits from the advantages derived from both analog and digital computers. An analog computer is a fully parallel device. For example, given the second order polynomial Y=A+B*X+C*X**2 (or Y=A+X*(B+C*X)), the dataflow diagram, seen in Fig. 1, shows that evaluating this expression requires two multiplications and two additions. Given that the equations to be solved require two multiplications and two additions, as seen in Fig 2A,

an analog circuit employs two multipliers, 22 and 26, and two adders, 20 and 24. While this approach offers the ultimate in speed, it also becomes expensive for large applications.

In contrast, a traditional digital computer employs a single arithmetic unit which is used repeatedly -- as many times as the equations (or program) may require. In the example given above, the same arithmetic unit is used four times to perform the desired calculation.

Direct replacement of analog components by their digital counterparts yields a "fully-parallel" digital system, with one component per mathematical operation as seen in Fig. 2B -- the adders are 30 and 34, the multipliers are 32 and 36, and the analog patch panel 28 from Fig. 2A is replaced with a digital switch network and memory 38. Although this digital architecture is capable of the fastest possible speed, it suffers from the same cost disadvantages as its analog predecessor.

Furthermore, such a digitally implemented analog architecture is time-wise inefficient because most of the components are idle most of the time. As mentioned, a fully parallel implementation requires two multipliers and two adders; but, there is no way all four components can operate simultaneously.

The dataflow diagram in Fig. 1 reveals that the first multiplication must finish before the first add can begin, and this must finish before the second multiplication can begin, and so on. In other words, regardless of the hardware configuration, the time required to evaluate this expression is at least two add times plus two multiply times. This limitation is known as "precedence constraints" or "data precedence". And it is this concept of precendence constraints or critical

path which limits the amount of parallelism available in an application.

Viewed from another perspective, it is the concept of precedence constraints which also creates an opportunity for economizing hardware by re-using components. Referring back to the configuration in Fig. 2B, given the above-mentioned precedence constraints, the same adder 30 and the same multiplier 32 can be used for each of the two operations as seen in Fig. 2C. In fact, given a component that could perform a host of operations including an add and multiply, the entire equation could be solved with one component such as an ALU 39 as seen Fig. 2D.

The above evolution (illustrated in Figs. 2A through 2D) merges the attractive aspects of both the analog and digital world.

Overall System Architecture

Fig. 3 shows a high-level block diagram of the exemplary embodiment of a system which incorporates the present invention. A system user accesses the system via a workstation 40. A host computer 42 compiles source code, written in CSSL, which is a widely-used language for programming continuous system simulations. CSSL is a simulation language standard of the Society for Computer Simulation. Since CSSL is a parallel language for describing parallel physical systems, it is well-suited for programming a parallel machine like the present invention.

The compiler, using the parallel CSSL code, schedules the instructions to be run on the available arithmetic computational modules (ACM's) 48 in the system. The method of scheduling is itself an aspect of the present invention which is discussed in detail below.

Then host 42 loads the information directly into the setup interface 44. It should be noted that a system incorporating the present invention is not a stand-alone computer system. It can be characterized as a peripheral system to a conventional host for which the host is needed for preparatory functions prior to running simulations on the system.

Next, setup interface 44 via bus 56a-c loads controller 46, ACM's 48 and I/O interfaces 50 with their respective instructions. The actual running of the instructions is controlled by a single program counter which resides in the controller 46. The controller 46 contains a program counter, an address stack, an instruction memory and register, and miscellaneous control logic. Although branching is relatively rare in a system incorporating the present invention, if a programmed application requires that it happen, the controller 46 by its design can accommodate this need.

Although Fig. 3 shows a single bus, designated 56a-c, connecting the setup interface 44, the controller 46, and the ACM's 48, it is actually three different busses. The first bus 56a is for setup/diagnostic purposes which connects the setup interface to the controller 46, to the ACM's 48 and to the I/O interfaces 50. It is this bus 56a which is used to load the individual instruction memories and data memories for the controller 46, the ACM's 48 and the I/O interfaces 50. Besides loading instructions, this bus 56a is also used for conducting system diagnostics.

The second bus 56b is a data bus which is used for data transfer among the ACM's 48 and the I/O interfaces 50. This bus 56b is much faster than the setup bus because the speed of run time is more crucial than the speed of setup/diagnostic time.

And, the third bus 56c is the program counter bus on which the controller 46 broadcasts the program count to each of the ACM's 48 and I/O interfaces 50 in the system. Again, because there is only one program counter for all of the active processors, ACM's 48 and I/O interfaces 50, the system is not well-suited for handling branches. This architectural decision was conscious and a result of balancing of the design's gained efficiency in the parallel processing of scalar operations versus the drawbacks of not branching easily. Note that an obvious branch that the system must be able to handle is the end of program branch -- at the end of a program the system must be able to start back at the beginning.

In addition to the ACM's 48 seen in Fig. 3, the system contains I/O interface modules 50. The I/O interface modules 50 are also connected to each of the three busses 56a, 56b and 56c for receiving instructions, transferring data, and accessing the program count, respectively. These modules are for digital and analog interfaces and are controlled by the same program counter that controls the ACM's 48. It is important that the computational processors, ACM's 48, as well as the I/O processors 50, for the present invention be controlled by the same program counter because the applications targetted by this system require that data computations and data transfers be time deterministic. And, for the compiler/scheduler to not only take maximum advantage of the fine-grained parallelism but also resolve potential bus conflicts prior to a program's execution, it must know exactly when such computations and transfers will occur.

An example of how the I/O interface 50 may be used is for "hardware-in-the-loop" simulations. These are real-time simulations where portions of the simulation

are carried out using a simulator, such as the present invention, and the remaining portions are carried out with the pieces the actual device(s) being simulated. The data being transfered between the simulator and the actual device(s) must be speedy, accurate, and precisely timed. Because the present invention compiles/schedules instruction execution and data transfer prior to a program run, it can easily provide the precise timing.

The remaining block of Fig. 3 is the run-time interface 52. The function of the run-time interface 52 is to allow data to be accessed and subsequently displayed on a scope or screen while simulations are taking place.

ACM Architecture

Fig. 4 shows a high-level functional block diagram of an ACM 48 consisting of an input register 62 and an output FIFO 76 both coupled to bus 56b for receiving and sending data, respectively, external to ACM 48. Also included in the ACM 48 is an indexed memory 66 and two operand memories 68a and 68b which provide the data inputs to ALU 72. In the exemplary embodiment of the present invention, the indexed memory is a 128Kx32 RAM and the operand memories 68a and 68b each include a pair of 1Kx32 RAMs (see Fig. 6B). The pair of 1Kx32 RAMs acts like a dedicated dual-port memory by allowing two values to be fetched from two of the RAMs and two values to be stored in the other two RAMs (a total of four memory references) per clock cycle.

The data outputs from input register 62, indexed memory 66 and ALU 72 are fed into a digital switch network 64, consisting of a group of strategically placed multiplexers, which provides flexible, internal dataflow trafficking within the ACM 48. The switch network 64, in

turn, supplies data inputs to the output FIFO 76 (for outputting to the bus 56b), the indexed memory 66 (for indexing functions) and the operand memories 68a and 68b (for subsequently supplying operands to ALU 72). As indicated by the arrows, all of the above functional blocks in the ACM 48 are controlled by a wide instruction word represented by block 74 which is actually an instruction memory and register.

Note that some of the functional blocks shown in Fig. 4 are labelled with their primary function and the additional descriptive phrase "AND LOGIC". This is done to better correlate this level diagram with the lower-level diagram discussed below.

Individual Functions: SWITCH, COMPARE, MAX/MIN

Fig. 5 shows additional details of the exemplary embodiment of an ACM 48. Fig. 5 shows two pairs of RAMs 88a, 88b and 90a, 90b, respectively. In the exemplary embodiment of the present invention, each of the RAMs 88a, 88b, 90a and 90b is a 1Kx32 RAM. The inputs to these RAMs, lines 102a and 104a, are provided by the switch network (although not shown in Fig. 5, it is element 64 in Fig. 4). The input to the respective pairs of RAMs 102a, 104a and the outputs of the RAMs are then fed to their respective muxes 92a and 92b. The outputs of the muxes 92a and 92b are then fed into registers 80a and 80b, respectively. The outputs of registers 80a and 80b are then fed to registers 82a and 82b as well as to ALU 86. The outputs of registers 82a and 82b and ALU 86 are then fed to mux 84.

It should be noted that the dotted lines indicate to which functional block, from Fig. 4, that the additional details belong. For instance, register 80a and 80b reside in the operand memory functional blocks 68a and

68b, respectively; while registers 82a, 82b, multiplexer 84 and ALU 86 reside in ALU functional block 72.

It should also be noted, as seen in Fig. 5, that mux 84 also has a "1" and a "0" input such that a "1" or a "0" can be generated by any function which may need these values as outputs. For example, a hardware comparator function (COMPARE) with data output as opposed to condition code output.

The COMPARE function is performed by loading the values to be compared in registers 80a and 80b. ALU 86 is instructed, via the instruction register (shown in Fig. 4 and Figs. 6A and 6B), to perform one of the compare operators (e.g., <, >, =, etc.). The typical output of this operation is a condition code from ALU 86, via line 83, which, in the exemplary embodiment of the present invention, is used as part of the mux select for mux 84 instead of feeding branching logic as in traditional architectures. The condition code, now part of the mux select, then selects either the "1" or "0" input to mux 84 depending on the result of the operation. Thus, the result is a hardware comparator function which produces a data formatted output rather than a condition code formatted output. It should be noted that the hard wired "1" and "0" are data formatted representations of the logical variables. In the exemplary embodiment of the present invention, the data format comprises a thirty-two bit data line, in which the dard wired "1" is represented by thirty-two "1"s and the hard wired "0" is represented by thirty-two "0"s.

Another function that ACM 48 performs in hardware is a SWITCH function. This too is efficiently implemented using the four registers 80a, 80b, 82a, and 82b and the multiplexer 84 surrounding the ALU 86. The SWITCH function has been designed to take the place of certain

conditional branches, for example, an IF-THEN-ELSE statement. For a system incorporating the present invention, this is most efficiently accomplished in hardware. This added functionality is necessary, as mentioned, because the present system is not well-suited to handle branching and ALU 86 does not provide this type of functionality. To explain how this hardware implements a SWITCH, a simple IF-THEN-ELSE example is used.

Given the statement

IF X THEN

Y = A,

ELSE

Y = Z

(which is represented in SWITCH form as

Y = SWITCH(X,A,Z),

on the first cycle the values A and Z are loaded into the first pair of registers, 80a and 80b. On the next cycle, X and 0 are loaded into the first pair of registers, 80a and 80b, and A and Z are shifted to the second pair of register, 82a and 82b. (It should be noted that the variable X could have been X=W<0 which was processed by a COMPARE function prior to being used by the SWITCH). At the same time, the ALU 86 is programmed by the instruction register (not shown) to perform the greater than (>) operation on its held inputs X and 0. The result of this compare is supplied by the control lines of the ALU 86 and, in turn, is used as a mux select for determining which value, A or Z, should be output from the mux 84. As mentioned, this configuration is necessary because the ALU 86 does not provide this functionality and the present system needs this

functionality to perform simulations effectively and efficiently.

Similar to the COMPARE and SWITCH functions are the MAX and MIN functions. The MAX function is described in detail from which an understanding of the MIN function is easily derived. Operation of the MAX function (e.g., MAX(X,Y)) is performed by loading registers 80a and 80b with the values for the variables X and Y. Next, ALU 86 is instructed to perform a greater than (">") function. Next, the condition code genrated by ALU 86, as was the case with the COMPARE and SWITCH functions, being part of the mux select, line 83, selects the greater of the two values contained in registers 82a and 82b.

Thus, by adding logic to ALU 86, an ACM 48 can perform not only the functions available in the ALU 86 but also the additional functions of COMPARE, SWITCH, MAX and MIN which make the ACM 48 and, consequently, the overall architecture faster, more efficient and better adapted for handling limited conditional branching functions.

Fig. 5 only provided a detailed diagram of a portion of ACM 48, a complete detailed diagram of ACM 48 is provided for the sake of completeness in Figs. 6A and 6B.

Beginning with Fig. 6A, the data input to ACM 48 via bus 56b is received by input register 62. The output of register 62 is fed into the switch network (element 64 in Fig. 4) shown as a group of four three-input muxes: muxes 100, 102 and 104 are shown in Fig. 6A and mux 106 is shown in Fig. 6B.

Also providing an input to the switch network is the indexed memory (element 66 in Fig. 4) shown in Fig. 6A as two parts. The first part which handles the index addressing comprises a RAM 110 which receives its input from the switch network. In the exemplary embodiment of

the present invention, RAM 110 is a 1Kx32 RAM. RAM 110 supplies one input of mux 112 where the other input is supplied directly from the switch network. The output of mux 112 is fed to register 114 which supplies one input to adder 116. The other input to adder 116 is supplied by the instruction register 113 which gets its input from the instruction memory 115 as seen in Fig. 6B. In the exemplary embodiment of the present invention, the instruction memory is a 16Kx128 RAM.

Adder 116 provides an address which is fed to the second part of the indexed memory which handles the storing of indexed data. The address is fed to a RAM 120. In the exemplary embodiment of the present invention, RAM 120 is a 128Kx32 RAM. RAM 120 supplies one input of mux 122 whose other input is supplied by the output of RAM 110. The output of mux 122 is registered by register 124 and fed into the switch network.

Fig. 6B shows many of the same details described in Fig. 5, thus Fig. 6B is briefly described. In addition to what has already been described in conjunction with either Fig. 5 or Fig. 6A, Fig. 6B shows the output of mux 84 feeding register 130. The output of register 130 then provides the final input to the switch network (this is represented by the output of register 130 feeding the third input to mux 106). Mux 106 is then fed to output FIFO 76 which delivers data to bus 56b. The output of mux 106 is also supplied to the LED register 132 which provides input to the LEDs 134.

Fig. 6B also shows the control outputs of ALU 86 supplied to registers 136 and 138. Additionally, the program count which is received from the single program counter (not shown), via bus 56c, is fed into the program count register 140. The output of register 140 is fed to

the instruction address register 142 which then provides it to the instruction memory 115.

Reviewing Figs. 6A and 6B, it is clear that because an ACM employs registers (e.g., 80a/80b, 82a/82b, 130, etc), the ACM is divided into stages or pipelines. And because the functionality of an ACM is pipelined, in general, a new instruction can begin with each new clock cycle. Pipelining effectively allows the ACM to process instructions in a parallel fashion which compounds the parallel processing capabilities of the entire architecture. In the exemplary embodiment of the present invention, the pipelining causes a typical latency of five cycles in the ACM.

Given this description of the hardware, how the software (the compiler) takes advantage of the parallel configuration of the programmable ACM's 48 is now described.

Compiler: Scheduling Function

To take full advantage of the parallel hardware configuration, the compiler breaks down the source code into a parallel equivalent that can be run on the hardware. Because the above-described architecture is designed for optimizing parallel scalar operations, in the exemplary embodiment of the present invention, the parallel equivalent is at the level of fine-grained parallelism.

Fine-grained parallelism means that the parallelism that exists at the scalar operator level is exploited. This results in a large number of elementary tasks. In the exemplary embodiment of the present invention, the elementary tasks which are supported by the architecture are one-input operators and two-input operators (an exception to this is the three-input SWITCH function).

Thus, the code is broken down into machine level tasks with one or two inputs.

However, because certain variables depend on other variables in a given program or simulation, optimizing via parallelism has its limitations. As explained with reference to the hardware, the controlling limitation is known as the critical path limit. The critical path limit is based on the order in which certain variables must be calculated. And, as previously mentioned, the requirement that one variable be calculated before another is known as "data precedence". It is this data precedence which defines the critical path limit and which cannot be improved upon by increasing parallel resources (i.e. putting more ACM's in the system).

For example, if Y=A*B+C+D then a typical compiler introduces a new variable such that X=A*B and Y=X+C+D. At this point, typical compilers continue to scan from left to right with the resulting equations looking like X=A*B, Z=X+C and Y=Z+D. Thus, a typical compiler, as seen in Fig. 7A, attempting to optimize the number of operations, calculates Y by first multiplying A and B then adding that the result to C and, finally, adding that result to D. This series of three operations results in the use of three sequential time slots. And assuming for comparative purposes that each time slot requires one system cycle, then the above example takes three (3) cycles. The above described aspects of compiling are well known in the art; however, the compiler for the present invention takes this optimizing a bit further.

The compiler aspect of the present invention recognizes that different variables are available at different times. The compiler of the present invention leaves the three-input sum as a three-input sum until

the entire source file is scanned and analyzed. Then the reduction is performed to minimize the critical path. For instance, the input variables C and D are available at the beginning because they are input variables. And, because they are objects of an add operation, they can be added during the same time that A and B are multiplied as seen in Fig. 7B. Subsequently, the results of the addition (C+D) and multiplication (A*B) are added to produce Y. In this version of calculating Y, because the addition and multiplication of the input variables occurs in parallel as seen in Fig. 7B, the computation only takes two sequential time slots. This translates into two system cycles. A speed-up of over 33%.

The way this is accomplished requires first that the source program be reduced into individual machine instructions. This is a straightforward operation currently performed by many compilers for conventional sequential machines. Each task consists of the calculation of a single output variable. In a dataflow diagram, such as seen in Fig. 1, each such variable may be thought of as the output of a "component", analogous to an analog component (Fig. 1 and 2a).

To obtain these two input tasks from the multiple input source code equations and then schedule them, the algorithm works by computing two separate values for each task. The first value is the earliest finish time (EFT) and the second value is the down-stream time (DST).

A listing of the source code which performs these functions is included as a microfiche appendix. The microfiche appendix includes two copyrighted 'C' programs: PHASE1.c and PHASE2.c. In the exemplary embodiment of the present invention, PHASE1.c performs the preprocessing of the source code (i.e., textual manipulation) and it also performs some of the EFT and

DST sorting and calculating; whereas, PHASE2.c performs the scheduling of scalar operations based on the preprocessing, EFT and DST information.

The EFT is the earliest time at which the task could be completed, given an unlimited number of processors and no bus contention. It is the length of the longest chain of dependent computations starting with state variables and constants, and leading to the completion of the task (or calculation of the output variable). The DST is the length of the longest path from the output of the component (the variable calculated by the task) to a "terminal" variable (one whose ouput is not needed as input for any further computations).

Fig. 8 shows a high-level flowchart illustrating how the scheduling function of the compiler works:

First, the EFT for each task is calculated as indicated by block 200. Next, the DST for each variable is calculated as indicated by block 204. Once the EFTs have been calculated, block 208, they are used to breakdown the multiple input operators by grouping operands according to their EFTs, block 206. Based on the availability of variables, the compiler then schedules tasks, block 210. The DST list, block 212, is used to resolve conflicts during the scheduling of tasks which are "ready" (due to the availability of operands) to be scheduled at the same time.

Calculating the EFT

As mentioned, the EFT for each task is calculated first. All state variables (e.g. integrator outputs) and constants have an EFT of zero by definition because they are available at the beginning of the step. Because these are the only variables available at the beginning of the step, any algebraic variable (one whose value

depends on the values of other variables) has an EFT determined by the EFTs of its input variables and the latency of the task generating it. For example, a multiplier task, represented in source form by Z=X*Y, has an EFT defined by EFT(Z) = MAX[EFT(X), EFT(Y)] + LATENCY where the LATENCY for a multiplier task on a system incorporating the present invention is five (5) cycles. This simply means that the multiplication can be finished, at the earliest, five (5) cycles after both inputs become available.

After setting all EFTs for state variables and constants to zero, all other EFTs are set to -1 indicating an undefined value. The algorithm then sweeps repeatedly through the tasks, looking for a task whose inputs have their EFTs defined, and applying the formula including the appropriate latency for each operation. During the sweep, the algorithm also sorts the variables in order of increasing EFT which guarantees that any variable that drives another variable, whether directly or indirectly, preceeds it on the list.

A first example of how the EFT is used to determine the most efficient breakdown follows. Suppose in the equation Y=A*B+C+D that A,B,C and D are all state variables. As mentioned, they would all have EFTs of zero. If the equation is broken down one level such that X=A*B and Y=X+C+D then the EFT of the intermediate variable X is five since the inputs have EFTs of zero and the operation has a latency of five cycles.

The desired output Y is the sum of three inputs, with EFTs of 5, 0 and 0. The scheduler now applies the rule: group together the operations with the smallest EFTs. Thus, C and D are grouped together, rather than X and C as was the case in the earlier example for typical compilers.

Now, the sum of C and D has an EFT of 5, and so does the product of A and B. Thus, the output Y has an EFT of 10 as compared with an EFT of 15 using the reduction practices of typical compilers which use the left to right scan of the source equation.

A second example assumes the EFTs for A, B, C and D are 5, 10, 15 and 20, respectively. In this case, the EFT of X= A*B is 15 (MAX(5,10)+5) and Y would be the sum of three inputs with EFTs of 15, 15 and 20. The present compiler would group X and C to produce an intermediate sum with an EFT of 20. The desired output Y is then calculated and has an EFT of 25.

To recapitulate, the optimum method of reduction depends on the entire dataflow graph describing the generation of A, B, C, D and Y. The single equation Y=A*B+C+D cannot be optimized in isolation. The present invention takes a global approach by reducing the entire source program to a single basic block (i.e., block without loops) which is possible because of the items described within such as macro expansion, loop unrolling, branch-free COMPARE and SWITCH operations, and others.

Calculating the DST

The next step is to calculate the DST for each task as indicated by block 204. This is done by sweeping backwards over the task list, in order of decreasing EFT. If a variable X drives a variable Z directly (as with the previous example), then DST(X) $\leq$ DST(Z) + LATENCY, where LATENCY is the latency for the operation generating Z. Note that, unlike the similar formula for EFTs, "$\leq$"is used rather than "=" since X may drive other variables with longer down stream paths. To perform this calculation, all DSTs are initially set to zero, and for each variable X that drives Z, the statement

```
IF ( DST(X) .LT. (DST(Z)+LATENCY) ) THEN
    DST(X)=DST(Z)+LATENCY
```

is executed. When this step is complete, all DSTs are correctly calculated. Note that this step requires only a single pass through the task list, since the variables are processed in order of decreasing EFT.

Scheduling

Next, a list is prepared of all variables that are "available", that is, whose values have been calculated. At the beginning, only state variables and constants are available. As tasks are scheduled, additional variables become available, and are added to the "available" list.

Any task whose input variables are all available is called "ready" task. Initially, only those tasks are ready whose inputs are either constants or state variables. Again, as the scheduling proceeds, additional variables become available and, consequently, additional tasks become ready and are added to the "ready list".

On any cycle, tasks are started on every processor for which there is at least one ready task. Such an algorithm is called a "greedy" algorithm because it never leaves a processor idle if there is work for it to do. Of course, it may happen that no task is ready, because all pending tasks are waiting for inputs which are in the process of being calculated in the pipelines. In that case, the processsor is forced to wait until the inputs are available.

Given that each processor starts some task if there are tasks ready for it, sometimes a choice needs to be made between tasks if there is more than one ready. If these circumstances arise, the algorithm chooses the task with the largest DST, since heuristically, it is the one

that is most likely to keep other tasks waiting later. If several tasks have equal DST, one is chosen arbitrarily (in the exemplary embodiment of the present invention, it is the one with the lowest identifying number).

Using this scheduling technique in conjunction with the above-described parallel architecture configuration, the system's performance is optimized.

Interleaving of Expanded Macros and Loop Unrolling

An additional feature of the present invention is the use of interleaving of macros. Because the exemplary embodiment of the present invention has a single program counter, it is difficult to handle branching. This is so because if the program counter were to encounter a branch, since it controls all of the ACM's in the system, it would cause all of the ACM's, as well as the I/O interfaces, to branch. And, if all of the ACM's and I/O interfaces in the system were to branch at the same time, many of them would be idle during the execution of the body of the branch. This is the type of inefficiency this aspect of the present invention is designed to overcome.

To reduce this inefficiency, the present invention employs the use of macros. The present invention uses macros to perform functions such as sine, cosine, etc. Instead of the entire system being idle during a subroutine call, the present invention expands the function macro into the necessary basic operations. Once that is complete, the compiler interleaves these basic operations with those operations needing to be run for the rest of the program, thus, the ACM's will not be unnecessarily idle. It should be noted that the use of

macro expansion is to allow for computational interleaving and not just to avoid linkage overhead.

Another type of expansion used by the present invention occurs during searches. As is well known in the art, one of the most efficient searching techniques for sequential data is the binary search. A binary search involves comparing a value, first, against the middle value of the sequence of data to be searched. Based on the comparison, either the first or second half of sequence can be eliminated. The same approach is repeated for the remaining half of the data and so on until the value is found or all data has been exhausted. The number of searches needed to be done is approximately the logarithm base 2 of the total number of data values. Knowing this in advance, the present invention simply copies the body of the search code that many times (log base 2 of number of data values) thus effectively "unrolling the loop". The unrolling of the loop creates a sequential program with no branches; in this form, the program is easily run on a system incorporating the present invention.

Miscellaneous: Parts List and Applications

All the circuits described above have been implemented using off the shelf parts. The simplier circuits such as registers and muxes abound in the market. In Figs. 5, 6A and 6B, the registers, multiplexers, ALU and memories are conventional elements. For example, the ALU 86 may be a TI8847. The workstation and host computer may be an IBM PC/AT. And the compiler which performs the scheduling function is written in 'C'.

Some specific applications which impacted the design of the present invention and for which, as expected, the present invention performs exceptionally well include: 1)

a six-degree of freedom (DOF) missile simulation, 2) a main rocket engine of a space shuttle simulation, 3) four-DOF robotic manipulator arm simulation, and 4) a small, but very stiff, chemical kinetics application.

Although the invention is illustrated and described herein embodied as a multiprocessor computer system designed for improved performance in the area of parallel scalar operations, the invention is nevertheless not intended to be limited to the details shown. Rather, various modifications may be made in the details within the scope and range of equivalents of the claims and without departing from the spirit of the invention.

What is Claimed:

1. A multiprocessor computer system having bus means for parallel processing of scalar operations, said system comprising:

a) single program counter means for sequencing through the addresses of a program,

b) a plurality of arithmetic computational modules (ACM's) each coupled to the single program counter means, each ACM comprising:

i) an instruction memory responsive to the single program counter means for providing a scalar operation instruction,

ii) means for providing, for each input to each ACM, an independent operand on each clock cycle,

iii) calculation means for performing, in accordance with the scalar operation instruction, a scalar operation on the operands and producing a data output,

iv) selecting means for receiving operands and a logical variable to select between operands as a function of the logical variable to provide a data-formatted output, and

v) switch network means for receiving the data-formatted output from the selecting means and routing it to the bus means.

2. The multiprocessor computer system of claim 1 in which the calculation means includes an arithmetic logic unit for comparing the operands and producing a data output instead of, or in addition to, a condition code, and

the selecting means including (1) means for producing at least two data-formatted variables and (2) multiplexer means for selecting between the data formatted variables as a function of the logical variable to provide the data-formatted output.

3. The multiprocessor computer system of claim 1 in which the calculation means includes an arithmetic logic unit coupled to first and second operand storage means for comparing the operands in the first operand storage means and for producing the logical variable, and

the selecting means including multiplexer means for selecting between the operands in the second operand storage means as a function of the logical variable to provide the data-formatted output.

4. The multiprocessor computer system of claim 1 in which each means for providing operands includes first and second operand memory means coupled to the instruction memory, said instruction memory including means for addressing the first and second operand memory means, wherein one operand memory means receives an operand and the other operand memory means provides an operand, respectively, on each clock cycle.

5. The multiprocessor computer system of claim 4 in which each of said first and second operand memory means includes two pairs of operand memories in which each pair performs simultaneous READ and WRITE operations, respectively, whereby on each clock cycle, a total of two operands are received and two operands are provided.

6. The multiprocessor computer system of claim 1 wherein said switch network means routes the data-formatted output as an operand to the means for providing operands for subsequent scalar operations.

7. The multiprocessor computer system of claim 1 which further comprises at least one interface processor means coupled to the single program counter means for transfering operands from an external source to the bus means.

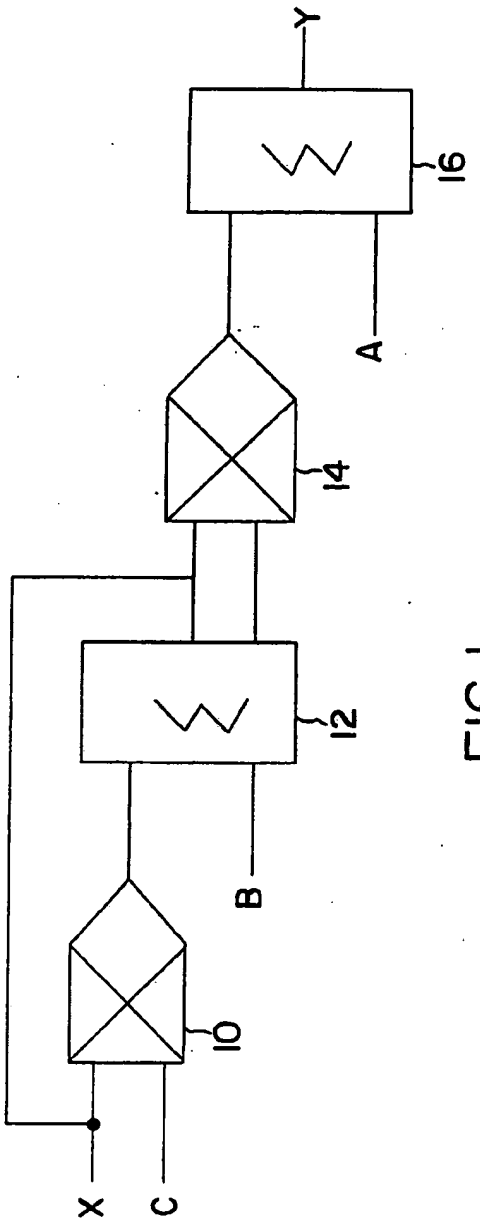


FIG. 1
PRIOR ART

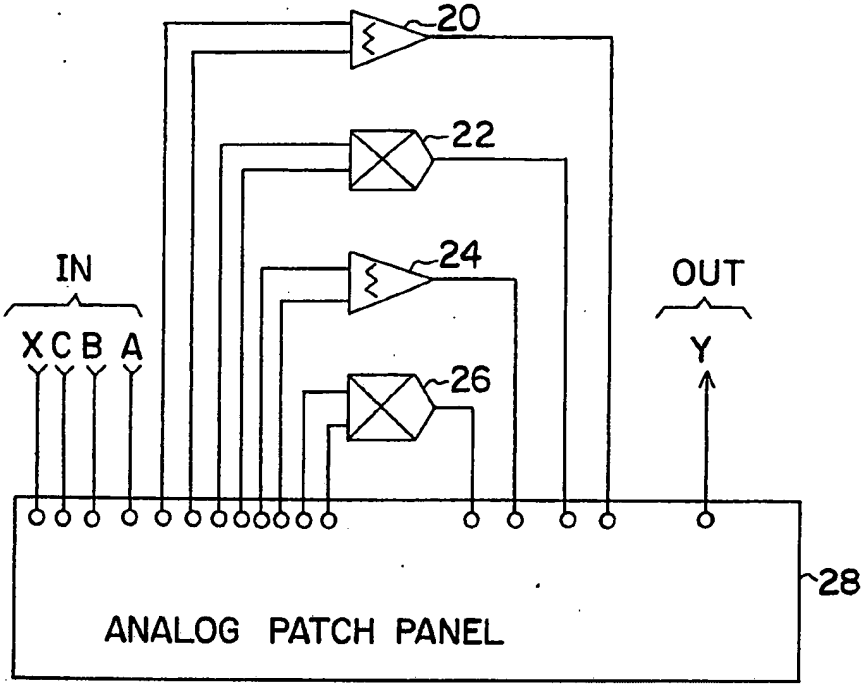


FIG.2A
PRIOR ART

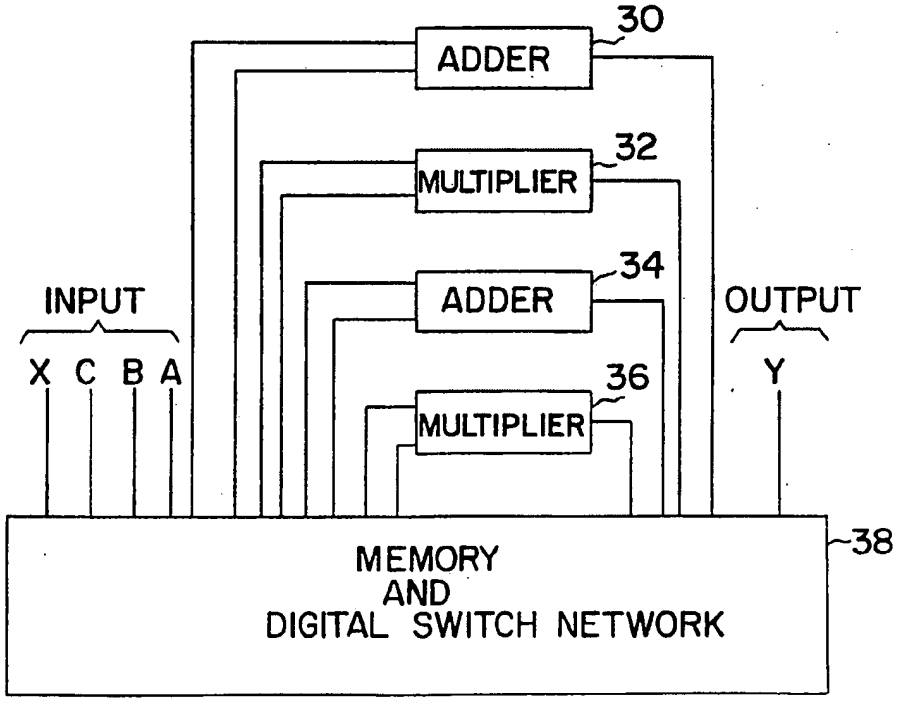


FIG.2B

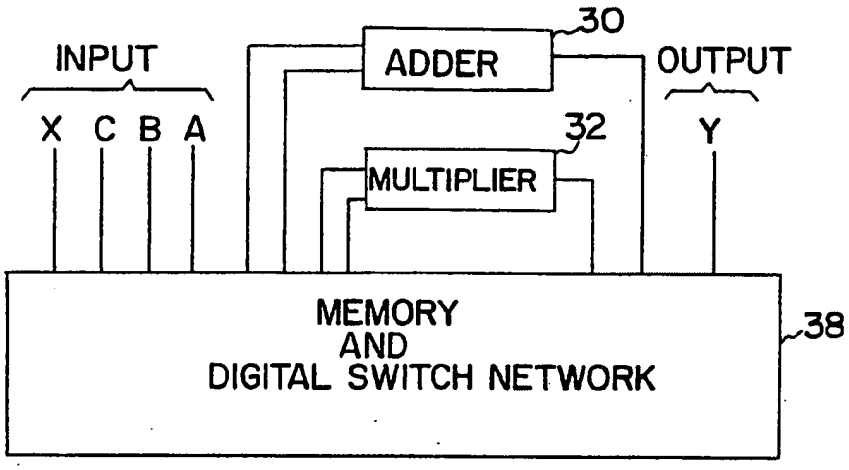

INPUT
X C B A
ADDER
30
32
MULTIPLIER
OUTPUT
Y
MEMORY
AND
DIGITAL SWITCH NETWORK
38

FIG.2C

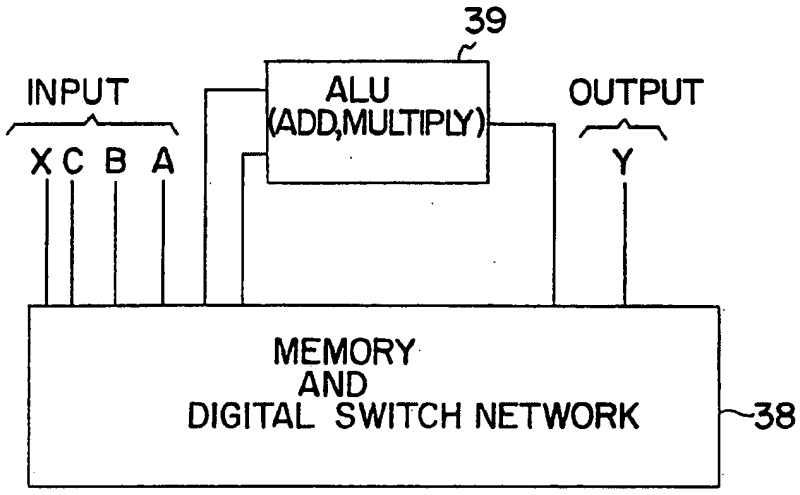

39
INPUT
X C B A
ALU
(ADD,MULTIPLY)
OUTPUT
Y
MEMORY
AND
DIGITAL SWITCH NETWORK
38

FIG.2D

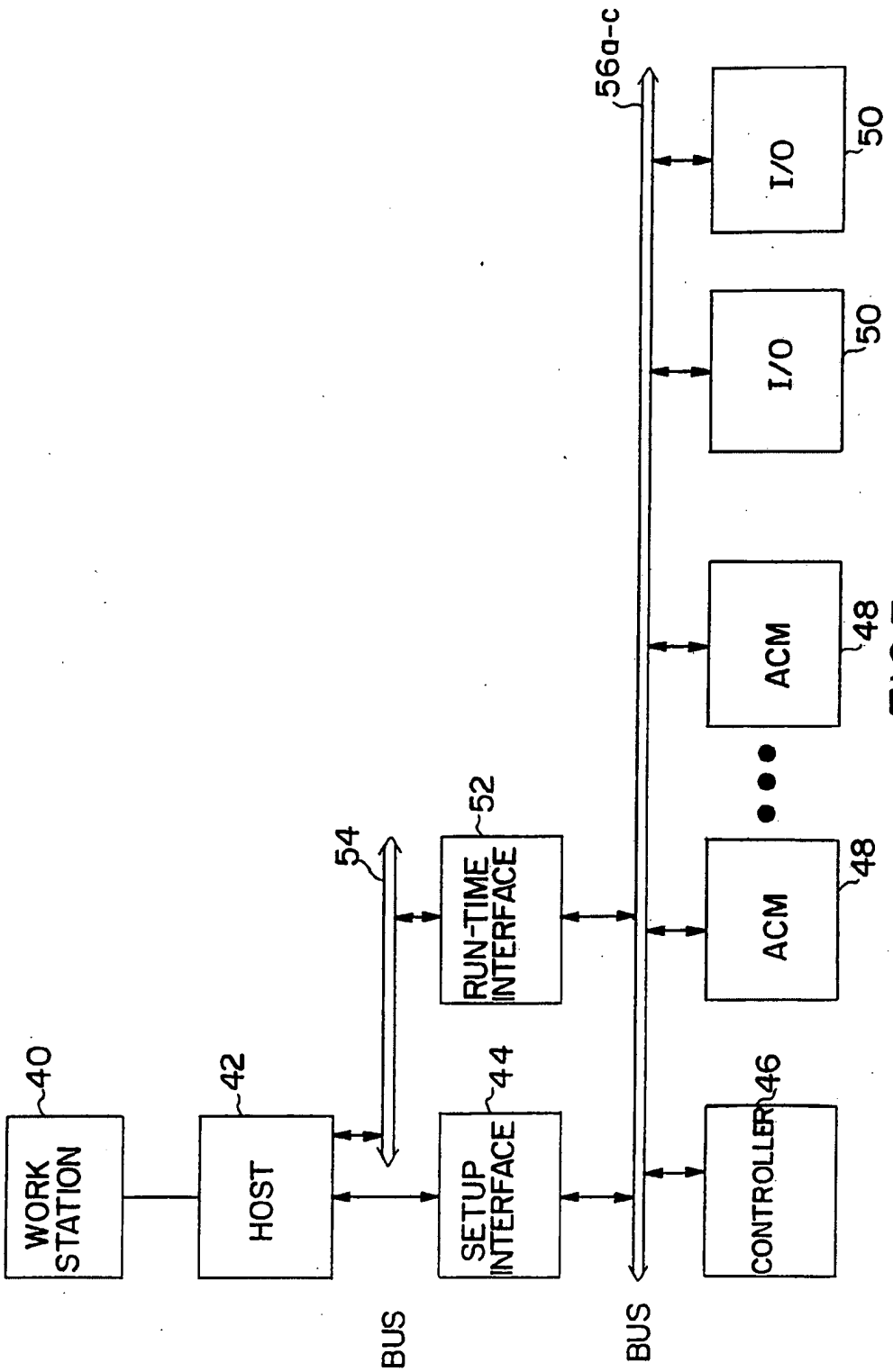

WORK STATION
40
HOST
42
SETUP INTERFACE
44
CONTROLLER
46
ACM
48
ACM
48
RUN-TIME INTERFACE
52
I/O
50
I/O
50
54
56a-c
BUS
BUS

FIG.3

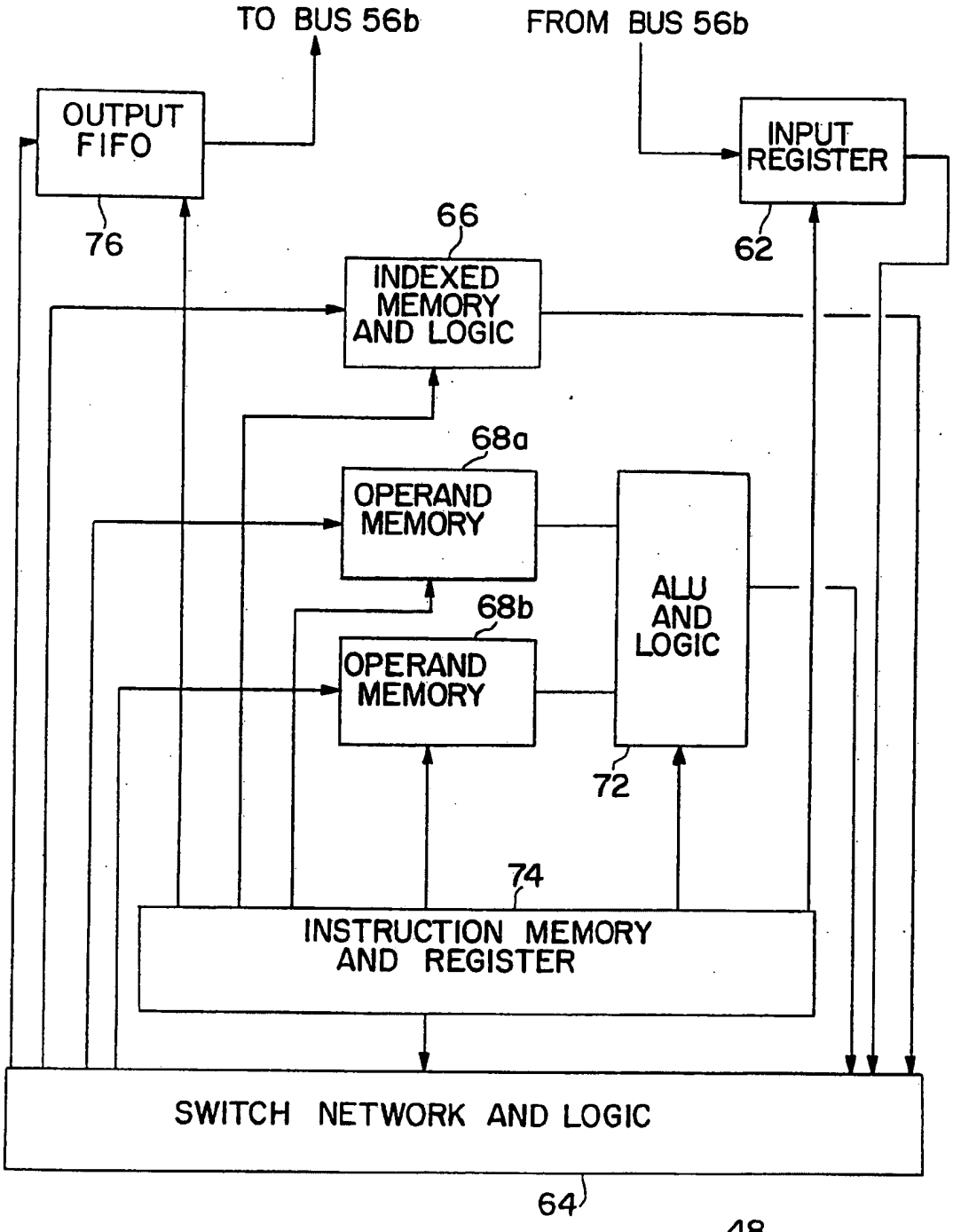

TO BUS 56b
FROM BUS 56b
OUTPUT FIFO
INPUT REGISTER
76
62
66
INDEXED MEMORY AND LOGIC
68a
OPERAND MEMORY
68b
OPERAND MEMORY
ALU AND LOGIC
72
74
INSTRUCTION MEMORY AND REGISTER
SWITCH NETWORK AND LOGIC
64
48

FIG.4

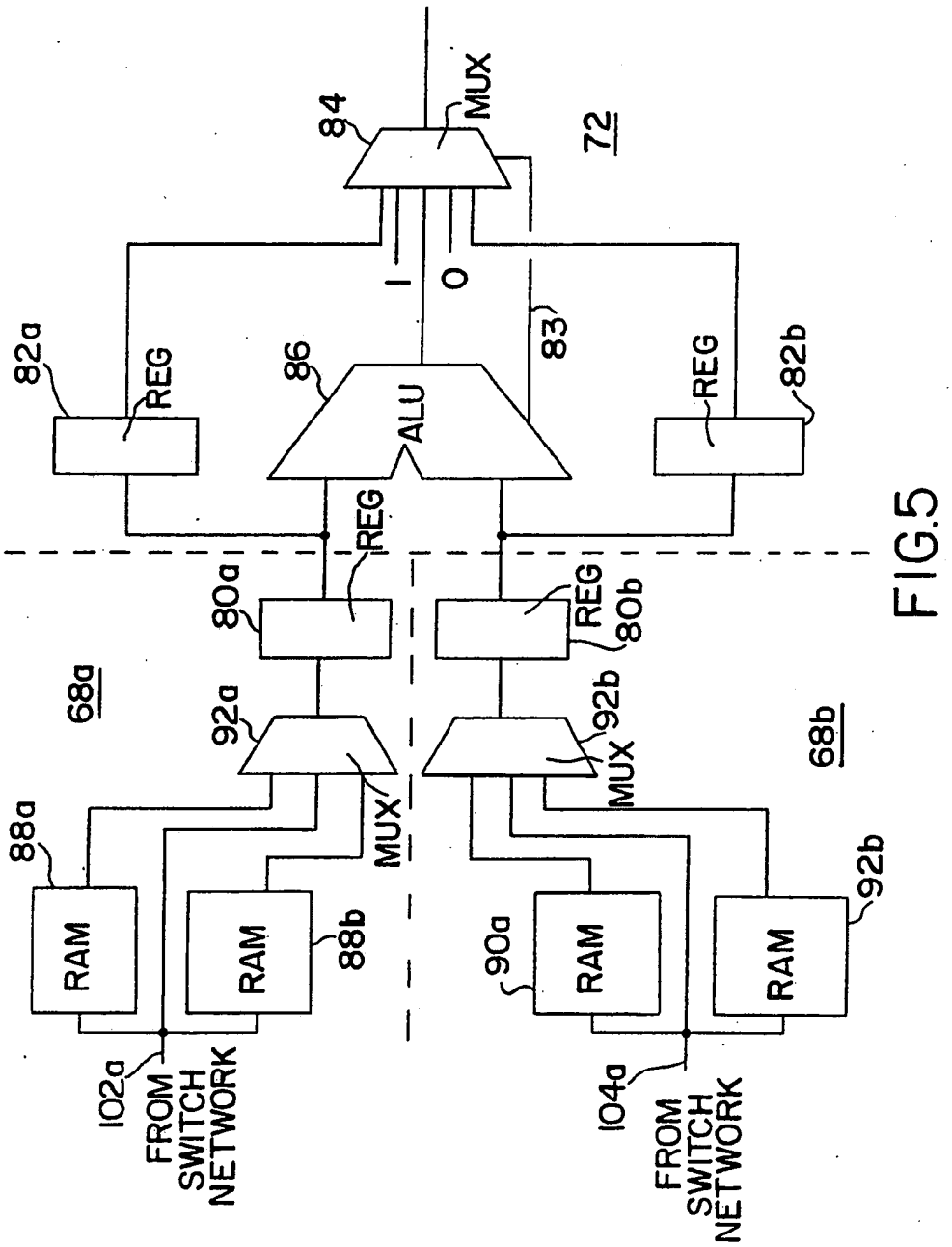

84
MUX
72
I
O
83
82a
REG
86
ALU
REG
82b
REG
80a
REG
80b
68a
92a
92b
MUX
MUX
68b
88a
RAM
RAM
88b
90a
RAM
RAM
92b
102a
FROM SWITCH NETWORK
104a
FROM SWITCH NETWORK

FIG.5

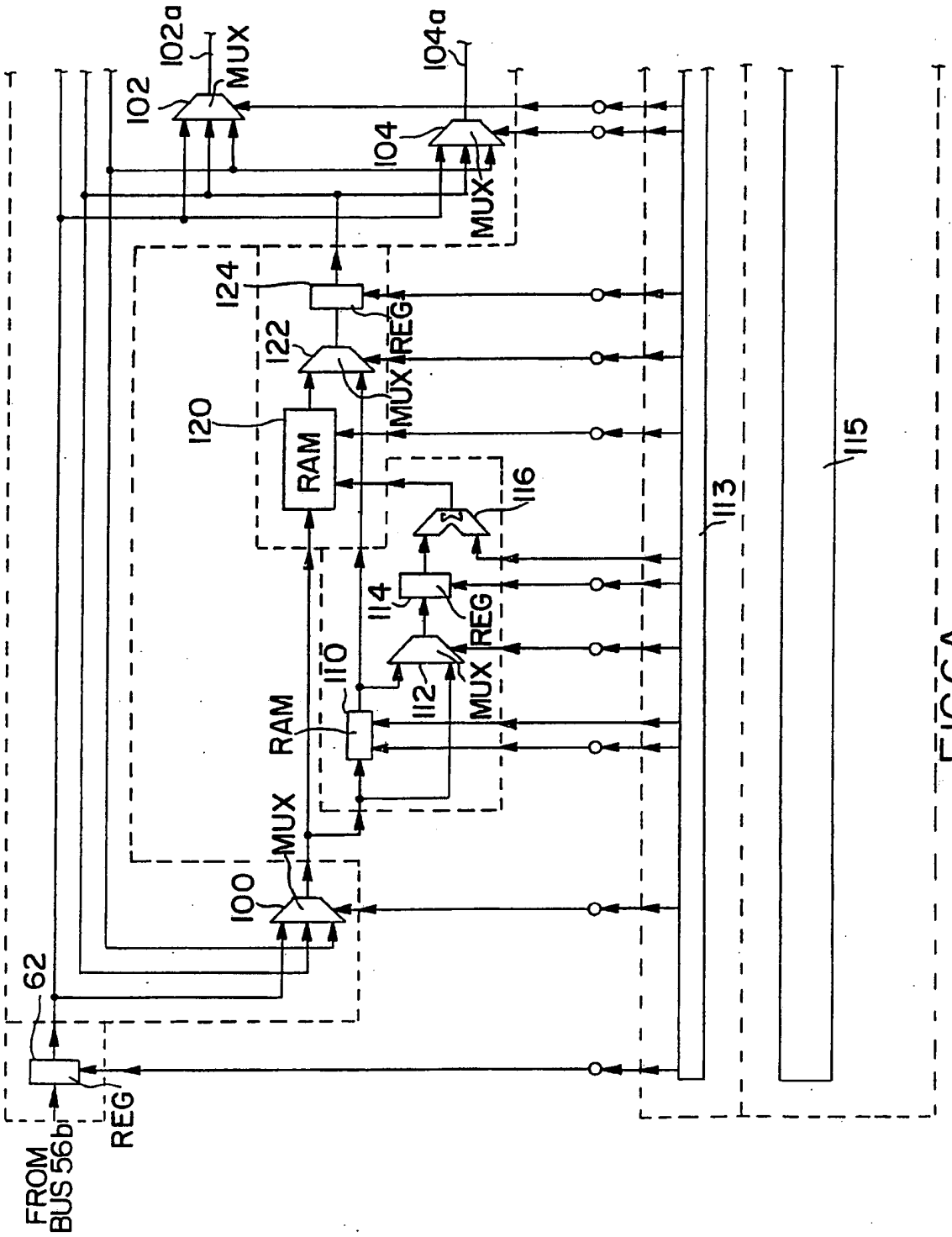

102a
MUX
102
104a
104
MUX
124
REG
122
MUX
120
RAM
116
114
REG
110
112
MUX
RAM
MUX
100
62
REG
FROM BUS 56b
113
115

FIG.6A

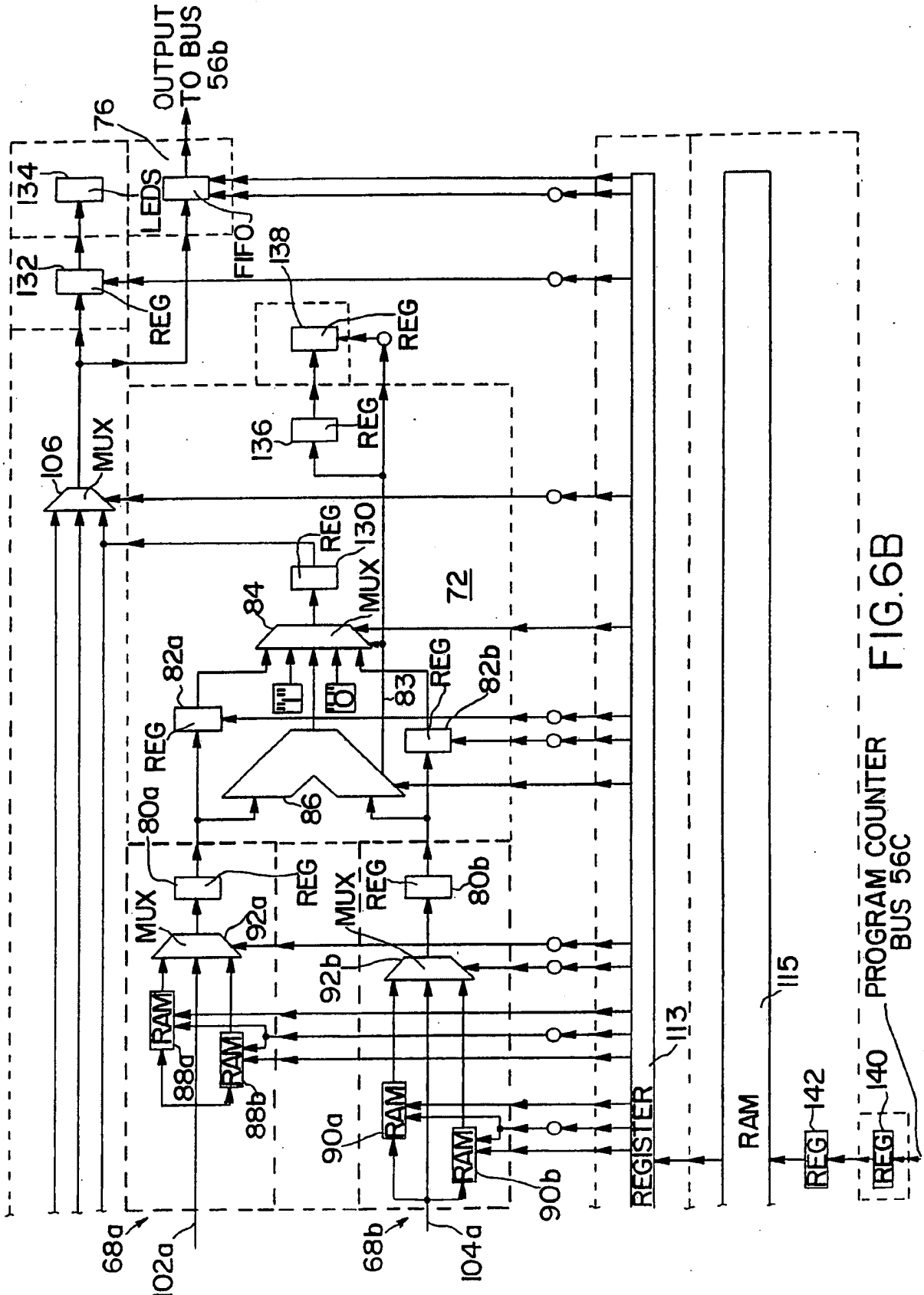

OUTPUT TO BUS 56b
76
134
LEDS
FIFO
132
REG
138
REG
136
REG
106
MUX
REG
130
MUX
72
84
82a
"1"
"0"
83
REG
82b
REG
80a
86
REG
MUX
REG
80b
MUX
92a
92b
RAM
88a
RAM
88b
90a
RAM
RAM
90b
REGISTER
113
115
RAM
142
REG
140
REG
PROGRAM COUNTER BUS 56C
68a
102a
68b
104a

FIG.6B

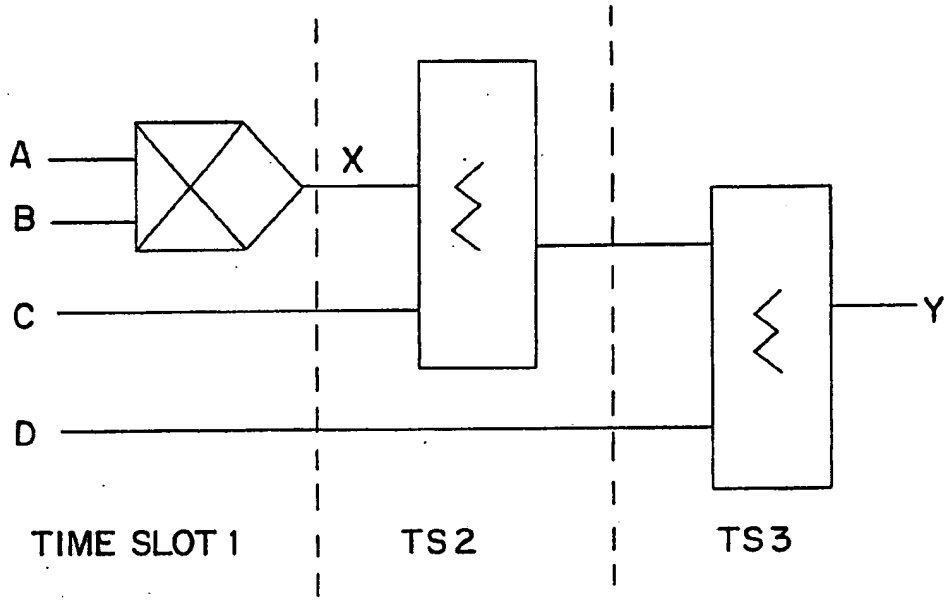


FIG.7A
PRIOR ART

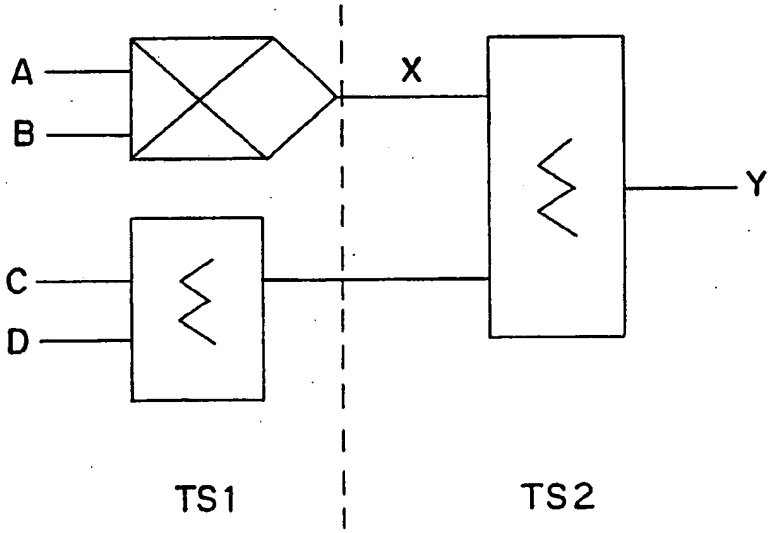


FIG.7B

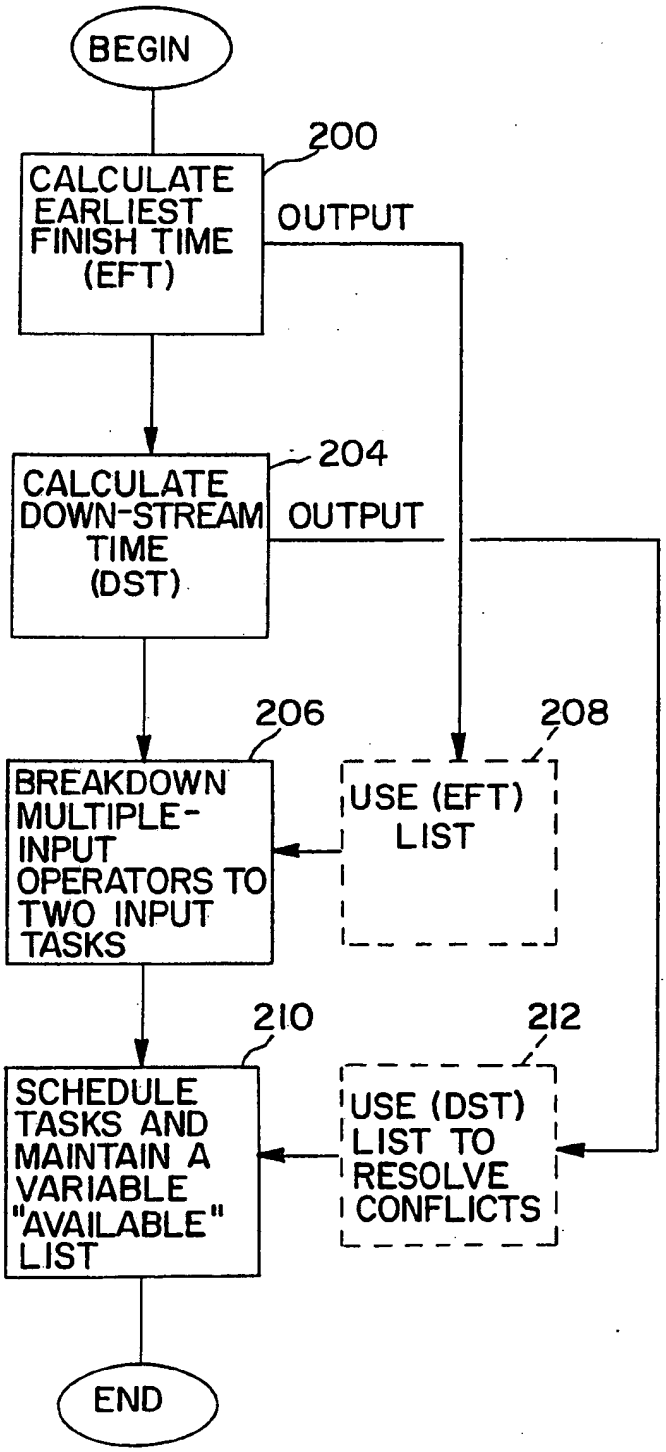

BEGIN
200
CALCULATE EARLIEST FINISH TIME (EFT)
OUTPUT
204
CALCULATE DOWN-STREAM TIME (DST)
OUTPUT
206
208
BREAKDOWN MULTIPLE-INPUT OPERATORS TO TWO INPUT TASKS
USE (EFT) LIST
210
212
SCHEDULE TASKS AND MAINTAIN A VARIABLE "AVAILABLE" LIST
USE (DST) LIST TO RESOLVE CONFLICTS
END

FIG.8

INTERNATIONAL SEARCH REPORT

PCT/US93/03165

A. CLASSIFICATION OF SUBJECT MATTER

IPC(5) :G06F 9/28 G06F 15/16
US CL :395/800,375; 364/133; 364/136; 364/130
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/800,375; 364/133; 364/136; 364/130

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS Database (USPAT file) searched.
Dialog Database searched.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US,A, 4,775,952 (Danielsson et al.) 04 October 1988 (See columns 3-7 and fig. 2).	1-7
P, X	US,A, 5,187,795 (Balmforth et al.) 16 February 1993 (columns 4-5, 12 and 22-23).	1-7
Y	US,A, 4,760,518 (Potash et al.) 26 July 1988 ((See figs (5a-b), 16-18) and columns 12-14 and 31-36).	1-7
Y	US,A, 4,996,661 (Cox et al.) 26 February 1991 (See columns 1-4).	1
Y	US,A, 5,023,826 (Patel) 11 June 1991 (See figures 3-5 and column 11-12)	1

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search	Date of mailing of the international search report
27 MAY 1993	19 JUL 1993

Name and mailing address of the ISA/US	Authorized officer
Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231	For MEHMET GECKIL Ngoc Nguyen NGUYEN NGOC-HO INTERNATIONAL DIVISION
Facsimile No. NOT APPLICABLE	Telephone No. (703) 305-9676

Form PCT/ISA/210 (second sheet)(July 1992)*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US93/03165

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US,A, 5,072,418 (Boutaud et al.) 10 December 1991 (See figures 1 and 1b and columns 6 and 11-16).	1-3